Table of Contents

Program Development Tools

Recommended Intel Compiler Debugging Options

DRAFT

This article is being reviewed for completeness and technical accuracy.

- Commonly used options for debugging:

  -O0
  : Disables optimizations. Default is -O2

  -g
  : Produces symbolic debug information in object file (implies -O0 when another optimization option is not explicitly set)

  -traceback
  : Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run-time.

    ```
    Specifying -traceback will increase the size of the
    executable program, but has no impact on run-time
    execution speeds.
    ```

  -check all
  : Checks for all run-time failures. **Fortran only.**

  -check bounds
  : Alternate syntax: -CB. Generates code to perform run-time checks on array subscript and character substring expressions. **Fortran only.**

    ```
    Once the program is debugged, omit this option to reduce
    executable program size and slightly improve run-time
    performance.
    ```

  -check uninit
  : Checks for uninitialized **scalar** varaibles without the SAVE attribute. **Fortran only.**

  -check-uninit
  : Enables run-time checking for uninitialized variables. If a variable is read before it is written, a run-time error routine will be called. Run-time checking of

undefined variables is only implemented on local, scalar variables. It is not implemented on dynamically allocated variables, extern variables or static variables. It is not implemented on structs, classes, unions or arrays. **C/C++ only**.

-ftrapuv
: Traps uninitialized variables by setting any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors. This option sets -g.

-debug all
: Enables debug information and control output of enhanced debug information. To use this option, you must also specify the -g option.

-gen-interfaces -warn interfaces
: Tells the compiler to generate an interface block for each routine in a source file; the interface block is then checked with -warn interfaces

- Options for handling floating-point exceptions:

-fpe{0|1|3}
: Allows some control over floating-point exception (divide by zero, overflow, invalid operation, underflow, denormalized number, positive infinity, negative infinity or a NaN) handling for the **main program** at run-time. **Fortran only.**

  - -fpe0: underflow gives 0.0; abort on other IEEE exceptions
  - -fpe3: produce NaN, signed infinities, and denormal results

  Default is -fpe3 with which all floating-point exceptions are disabled and floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero. Use of -fpe3 on IA-64 systems such as Columbia will slow run-time performance.

-fpe-all={0|1|3}
: Allows some control over floating-point exception handling for **each routine** in a program at run-time. Also sets -assume ieee_fpe_flags. Default is -fpe-all=3. **Fortran only.**

-assume ieee_fpe_flags
: Tells the compiler to save floating-point exception and status flags on routine entry and restore them on routine exit. This option can slow runtime performance. **Fortran only.**

-ftz

Flushes denormal results to zero when the application is in the gradual underflow mode. This option has effect only when compiling the **main program**. It may improve performance if the denormal values are not critical to your application's behavior. For IA-64 systems (such as Columbia), -O3 sets -ftz. For Intel 64 systems (such as Pleiades), every optimization option O level, except -O0, sets -ftz.

- Options for handling floating-point precision:

-mp
: Enables improved floating-point consistency during calculations. This option limits floating-point optimizations and maintains declared precision. -mp1 restricts floating-point precision to be closer to declared precision. It has some impact on speed, but less than the impact of -mp.

-fp-model precise
: Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations. These semantics ensure the accuracy of floating-point computations, but they may slow performance.

-fp-model strict
: Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations and enables floating-point exception semantics. This is the strictest floating-point model.

-fp-speculation=off
: Disables speculation of floating-point operations. Default is -fp-speculation=fast

-pc{64|80}
: For Intel EM64 only. Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the -pc option. -pc64 sets internal FPU precision to 53-bit significand. -pc80 is the default and it sets internal FPU precision to 64-bit significand.

Totalview

DRAFT

This article is being reviewed for completeness and technical accuracy.

TotalView is a GUI-based debugging tool that gives you control over processes and thread execution and visibility into program state and variables for C, C++ and Fortran applications. It also provides memory debugging to detect errors such as memory leaks, deadlocks and race conditions, etc.

Totalview allows you to debug serial, OpenMP, or MPI codes.

Totalview is available on both Pleiades and Columbia. See Totalview Debugging on Pleiades for some basic instructions on how to start using Totalview on Pleiades.

See Totalview Debugging on Columbia for some basic instructions on how to start using Totalview on Columbia.

Totalview Debugging on Pleiades

DRAFT

This article is being reviewed for completeness and technical accuracy.

TotalView is an advanced debugger for complex and parallel codes. Its versions have been installed as modules. To find out what versions of totalview are available, use the 'module avail' command.

There are additional steps needed in order to start the TotalView GUI. You'll need to rely on the ForwardX11 feature of your ssh. First, you'll have to make sure that your sysadmin had turned on ForwardX11 when SSH was installed on your system or use the -X or -Y (if available) options of ssh to enable X11 forwarding for your SSH session.

For debugging on a back-end node, do:

- Compile your code with -g
- Start a PBS session. For example:

```
% qsub -I -V -lselect=2:ncpus=8,walltime=1:00:00
```

- Test the X11 forwarding with xlock

```
% xclock
```

- Load the totalview module

```
% module load apps/etnus/totalview.8.6.2-1
```

- Set the environment variable TOTALVIEW

```
% setenv TOTALVIEW `which totalview` (for csh users)
```
or
```
% export TOTALVIEW=`which totalview` (for bash users)
```

- Start TotalView debugging
    - For serial applications:
        - Simply start totalview with your application as an argument

```
% totalview ./a.out
```

If your application requires arguments:

```
% totalview ./a.out -a app_arg_1 app_arg_2
```

- For MPI applications:

1. Make sure you load the appropriate modules, including the compiler, and mpi module. For example:

   For applications built with SGI's MPT, make sure that you have loaded the latest MPT module:

   ```
   % module load comp-intel/11.1.072
   % module load mpi-sgi/mpt.1.26
   ```

   For applications built with MVAPICH:

   ```
   % module load comp-intel/11.1.072
   % module load mpi-mvapich2/1.4.1/intel
   ```

2. Launch totalview by typing "totalview" all by itself. Once the totalview windows pop up, you will see four tabs in the "New Program" window: Program, Arguments, Standard I/O and Parallel.

3. Fill in the executable name in the "Program" box or use the Browse button to find the executable

4. Give any arguments to your executable by clicking on the "Arguments" tab and filling in what you need. If you need to redirect input from a file, do so by clicking the "Standard I/O" tab and filling in what you need.

5. In the "Parallel" tab, select the parallel system option MVAPICH2 or mpt_1.26 depending on which version of MPI you have compiled with.

6. Enter in the number of processes in the 'tasks' box; leave the 'nodes' field 0. For example, if you run your application with 2 nodes x 4 MPI processes/node = 8 processes in total, fill in 8 in the 'tasks' box and 0 in the 'node' box.

7. Then press "Go" to start. Note that it may initially dump you into the mpiexec assembler source which is not your own code.

8. Respond to the popup dialog box which says "Process xxx is a parallel job. Do you want to stop the job now?" Choose "No" if you just want to run your application. Choose "Yes" if you want to set breakpoint in your source code or do other tasks before running.

More information about TotalView can be found at the Totalview online documentation website.

Totalview Debugging on Columbia

DRAFT

This article is being reviewed for completeness and technical accuracy.

TotalView is an advanced debugger for complex and parallel codes. It has been installed as modules. To find out what versions of totalview are available, use the command 'module avail totalview'.

You'll need to rely on the ForwardX11 feature of your ssh. First, you'll have to make sure that your sysadmin had turned on ForwardX11 when SSH was installed on your local system or use the -X or -Y (if available) options of ssh to enable X11 forwarding for your SSH session.

For debugging on the front-end cfe2, do:

- Login to the front-end cfe2
- Compile your code with -g
- Make sure that X11 forwarding works and test it with xclock

```
cfe2%echo $DISPLAY
cfe2:xx.0
cfe2%xclock
```

- Load the totalview module

```
cfe2% module load totalview.8.9.0-1
```

- Start totalview. For serial jobs:

  cfe2% totalview a.out

  For MPI jobs built with SGI's MPT library:

  cfe2% totalview mpirun.real -a -np xxx a.out

For debugging on a back-end node, do:

- Compile your code with -g
- Start a PBS session and pass in the environment variable DISPLAY. Assuming PBS assign your job to run on Columbia21

```
cfe2% qsub -I -v DISPLAY -lncpus=8,walltime=1:00:00
```

- Test the X11 forwarding with xlock

```
PBS(8cpus)columbia21% xclock
```

- Load the totalview module

```
PBS(8cpus)columbia21% module load totalview.8.9.0-1
```

- Start totalview. For serial jobs:

  PBS(8cpus)columbia21% totalview a.out

  For MPI jobs built with SGI's MPT library:

  PBS(8cpus)columbia21% totalview mpirun.real -a -np xxx a.out

More information about TotalView can be found at the Totalview online documentation website.

IDB

DRAFT

This article is being reviewed for completeness and technical accuracy.

The Intel Debugger is a symbolic source code debugger that debugs programs compiled by the Intel Fortran and C/C++ Compiler, and the GNU compilers (gcc, g++).

IDB is included in the Intel compiler distribution. For IA-64 systems such as Columbia, both the Intel 10.x and 11.x compiler distributions provide only an IDB command-line interface. To use IDB on Columbia, load an Intel 10.x or 11.x compiler module. For example:

```
%module load intel-comp.11.1.072
%idb
(idb)
```

For Intel 64 systems such as Pleiades, a command-line interface is provided in the 10.x distribution and is invoked with the command *idb* just like on Columbia. For the Intel 11.x compilers, both a graphical user interface (GUI), which requires a Java Runtime, and a command-line interface are provided. The command *idb* invokes the GUI interface by default. To use the command-line interface under 11.x compilers, use the command *idbc*. For example:

```
%module load comp-intel/11.1.072 jvm/jre1.6.0_20
%idb
.... This will bring up an IDB GUI ....

%module load comp-intel/11.1.072
%idbc
(idb)
```

Be sure to compile your code with the *-g* option for symbolic debugging.

Depending on the Intel compiler distributions, the Intel Debugger can operate in either the gdb mode, dbx mode or idb mode. The available commands under these modes are different.

For information on IDB in the 10.x distribution, read **man idb**.

For information on IDB in the 11.x distribution, read documentations under *pfe or cfe2:/nasa/intel/Compiler/11.1/072/Documentation/en_US/idb*

GDB

DRAFT

This article is being reviewed for completeness and technical accuracy.

The GNU Debugger, GDB, is available on both Pleiades and Columbia under /usr/bin. It can be used to debug programs written in C, C++, Fortran and Modula-a.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Be sure to compile your code with *-g* for symbolic debugging.

GDB is typically used in the following ways:

- Start the debugger by itself
```
%gdb
(gdb)
```

- Start the debugger and specify the executable
```
%gdb your_executable
(gdb)
```

- Start the debugger, and specify the executable and core file
```
%gdb your_executable core-file
(gdb)
```

- Attach gdb to a running process
```
%gdb your_executable pid
(gdb)
```

At the prompt (gdb), type in commands such as *break* for setting a breakpoint, *run* for starting to run your executable, *step* for stepping into next line, etc. Read **man gdb** to learn more on using gdb.

Using pdsh_gdb for Debugging Pleiades PBS Jobs

DRAFT

This article is being reviewed for completeness and technical accuracy.

A script called *pdsh_gdb*, created by NAS staff Steve Heistand, is available on Pleiades under */u/scicon/tools/bin* for debugging PBS jobs **while the job is running**.

Launching this script from a Pleiades front-end node allows one to connect to each compute node of a PBS job and create a stack trace of each process. The aggregated stack trace from each process will be written to a user specified directory (by default, it is written to ~/tmp).

Here is an example of how to use this script:

```
pfe1% mkdir tmp
pfe1% /u/scicon/tools/bin/pdsh_gdb -j jobid -d tmp -s -u nas_username
```

More usage information can be found by launching pdsh_gdb without any option:

```
pfe1% /u/scicon/tools/bin/pdsh_gdb
```